Distributable Java Servlet Framework

A Senior Honors Project Presented

By

Joe Kislo

Submitted May 2000

APPROVED

Jack Wileden, Computer Science

Brian Levine, Computer Science

Mani Krishna, Electrical and Computer Engineering

Table of Contents

Introduction

Born from the first webservers was a standard called the Common Gateway Interface (CGI), which gave them the ability to serve dynamic content. Web-based server-side architecture has been slowly maturing since the introduction of the original NCSA webserver. The newest, and widely considered the best, architecture is the Java servlet. The servlet has some major differences from its most popular predecessor, CGI. These differences are the key that opens the door to some of the major advantages of servlets. This honors project will exploit these advancements in Java servlets to allow new designs which would have been previously impossible with CGI.

It is important to understand exactly what a CGI is before contrasting it with a servlet. This document refers to a CGI as an application that communicates with a webserver using the CGI standard. The CGI standard is very simple, and a CGI can be written in any executable language, including perl, Java, or C. The CGI standard dictates that when a web request comes to the webserver, the webserver will, in turn, start a new process and run the CGI. The webserver places the web request data inside the environment of that new process. Then the newly formed process initializes itself, reads the environment data, processes the request, and returns the output to the webserver and terminates. The webserver then sends the CGI's output to the user who made the request. If two requests come in simultaneously to the webserver, the webserver must still spawn and initialize two identical separate processes. It becomes clear that a CGI's lifecycle is very short, and a graphical representation can seen in **Figure 1.**

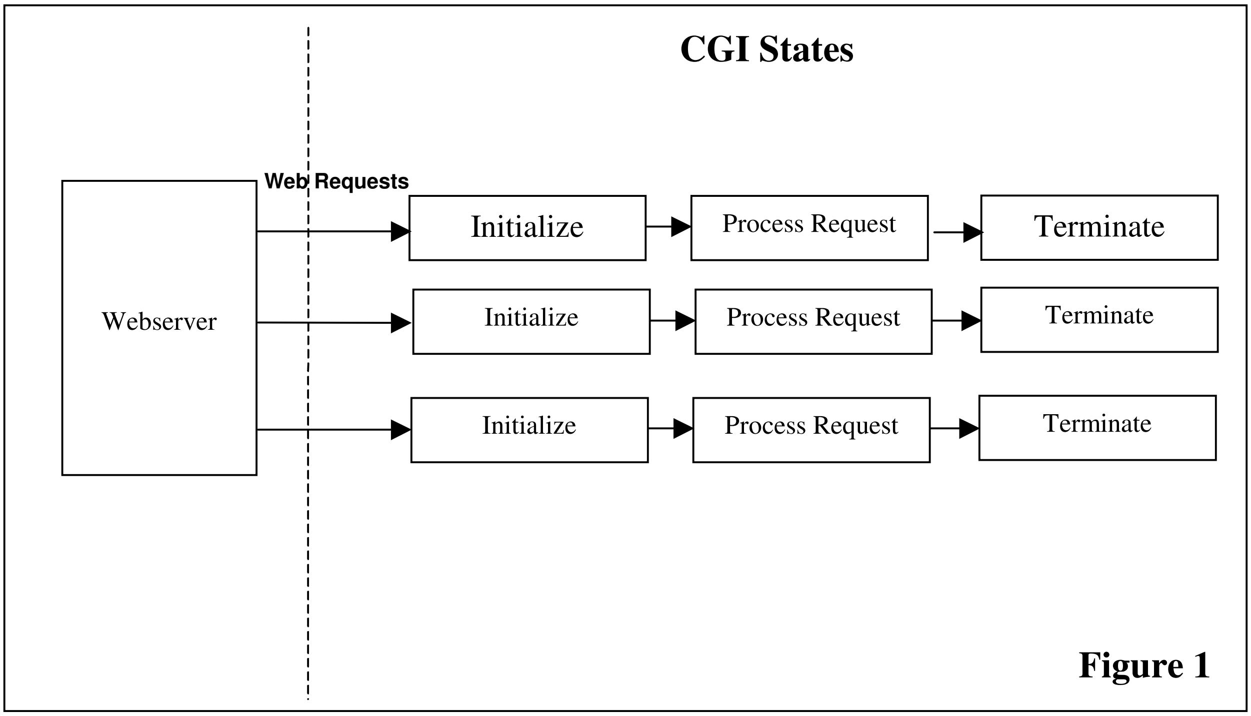
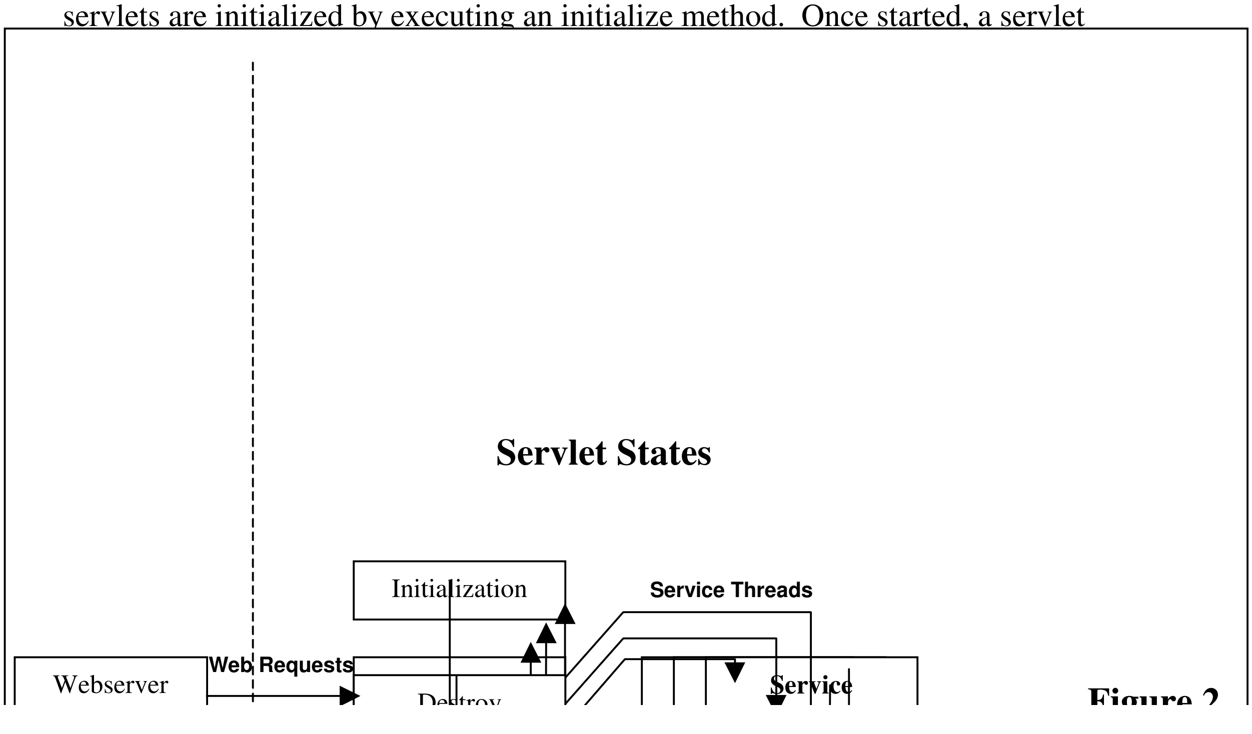


Figure 1

A servlet is a Java-only standard and, unlike CGI, a servlet cannot be written in any other language. Since a servlet must be written in Java, adding servlet support to a webserver usually means adding a Java Virtual Machine (JVM) to the webserver. Often a JVM is spawned for the webserver, and the two communicate using Interprocess Communications. All servlets for a particular website live and die inside the same JVM, which is the first major difference between a servlet and a CGI: all servlets attached to a single webserver reside inside a single process. The lifecycle of a servlet follows a very different path than a CGI. When the webserver is started, or at the very first request to a servlet, the servlets are initialized by executing an initialize method. Once started, a servlet



The mission of this project is to create a general set of services, known as a framework, that can be used with almost all servlet-based web applications. After examining the Java servlet, it's clear Sun Microsystems created an architecturally superior interface compared to CGI. However , Sun's definition of servlets doesn't specify how to actually write and structure an application. The services I have decided to create, and put into my framework, serve to enforce a structured approach to creating the web application. With the right tools, programming for the web can not only become easier, but can help to create a manageable and consistent codebase. The Page Manager facilities included in the framework, in particular, help to serve this function by forcing programmers to separate the functionality of the application into different Java packages.

This framework is intended to be an ideal programming environment for engineers, and also perform admirably when deployed and serving a site with a heavy load. Later in this report we discuss optimizing a website driven by this framework, including the possibility of using additional computers for load balancing. Other techniques and future enhancements for improving performance will also be discussed.

Services

The services in the framework can be categorized into the following groups:

Summary

Long Sessions

- **Session Manager** – The session manager serves to create and store a user's session between web requests. A user's session is a bucket that can contain any object pertaining to the state of the web user.

Content Management

- **Page Manager** – The page manager directs inbound web requests to the correct module inside the servlet. Modules are registered into a tree structure, allowing similar modules to be grouped together.
- **Template Manager** – Manages HTML content in the form of templates, which will later be modified from their template form and displayed to the user.

Logging Services

- **Log Manager** – Provides a standard interface for logging errors and exceptions, regardless of whether the output is going to a webserver log or being appended to a text log.

Load Balancing

- **Node Manager** – The node manager serves as the authority in the entire load balancing process. It will field requests for pointers to RemoteNodes which are capable of handling system load.
- **Node Scout** – The node scout lives inside the Node Manager as a daemonized thread. Its responsibilities are to actively seek nodes which are capable of handling system load, through the use of Java Jini services. The node scout will also determine each node's effective load capacity and services offered, and return this data to the Node Manager. The load-balancing algorithm, described later, is executed by the Node Scout to determine the optimal destination for a particular request.

These services will be created when the servlet is initialized, and will be available to the servlet throughout its life. A particular servlet may not need to make use of all these services. A detailed description of the services currently included in the framework can be found below.

Session Manager

The Session Manager is responsible for keeping state for a user between requests. Unlike traditional desktop programs, web programming does not naturally contain state. Web programming is also inherently multi-user. These two major differences bring forth the need for a session manager. Some simpler applications may not require these services, such as ones that are truly stateless. For those that require a login, upon login the session manager will assign a session object to that user. The session object acts as a bucket, and the programmer can place any object into it. This session object will be available whenever the user is in the system. For example, when a user logs into the system, the user may select what time zone they are currently in. The servlet could put this information into the session object, so it is easily available later. The session object can also be used for temporary variables and objects, such as a shopping cart.

The idea of a sessioning system is not new. The Java Servlet API actually specifies a sessioning system. The sessioning system the servlet API specifies works very similarly to my own system, with a few key differences. The major problem with using the built-in session system is that it is not extensible. The sessioning system is implemented by the webserver, built to match the Java Servlet API. However, since the sessioning system is webserver specific, web application developers can only count on things specified in the Java Servlet API to be true between different webservers. If they need to extend the sessioning system, you will have to extend the webserver's classes, thereby tying your servlet to a single webserver implementation. This negates some of the advantages of using Java for portable code.

One such limitation that the Java Servlet specification does not make clear is how sessions are stored when a servlet is stopped. This becomes a major concern, because a servlet is stopped and restarted automatically by the webserver whenever its class files are updated. If the session data is not stored during this process, it may result in logging out any users currently in the system. Some webservers have a setting for how many sessions will be stored on disk. Unfortunately, some do not have any facilities to store them at all, and all sessions are thrown away when the servlet restarts. This is unacceptable for most web applications. This is why a dedicated, webserver-independent, session manager was needed to replace the Java servlet sessioning system.

Page Manager

For large projects, modularity and maintainability become increasingly important. After I worked on the same web application codebase for over a year, it became clear that different parts of a web application needed to be separated into different modules or packages. Java already has a very elaborate package system, where class files associated with a package are stored in separate directories. It made sense to use this facility to separate the different modules. The Page Manager is responsible for delivering an incoming web request to the correct module. The web request is examined for its destination module, and the request is then passed on to that module.

A distributed registration system is used to program the Page Manager with the modules available. This is described as a tree structure, which allows related modules to be grouped together. There is an interface, called a **PageHandler**, for objects which have the ability to handle an incoming web request. An abstract class named **PageBroker**, which implements a PageHandler, is extended for a node on the tree which has the ability to have children. A tree structure is used instead of a flat structure so the tree can be built by parents registering children. This means a parent will register all the nodes it will require to function, and not run the risk of a parent node not registering a required node. A graphical example of a sample tree structure is shown in **Figure 3**. In this case, the Admin module has three submodules, which serve specific functions for the Admin module.

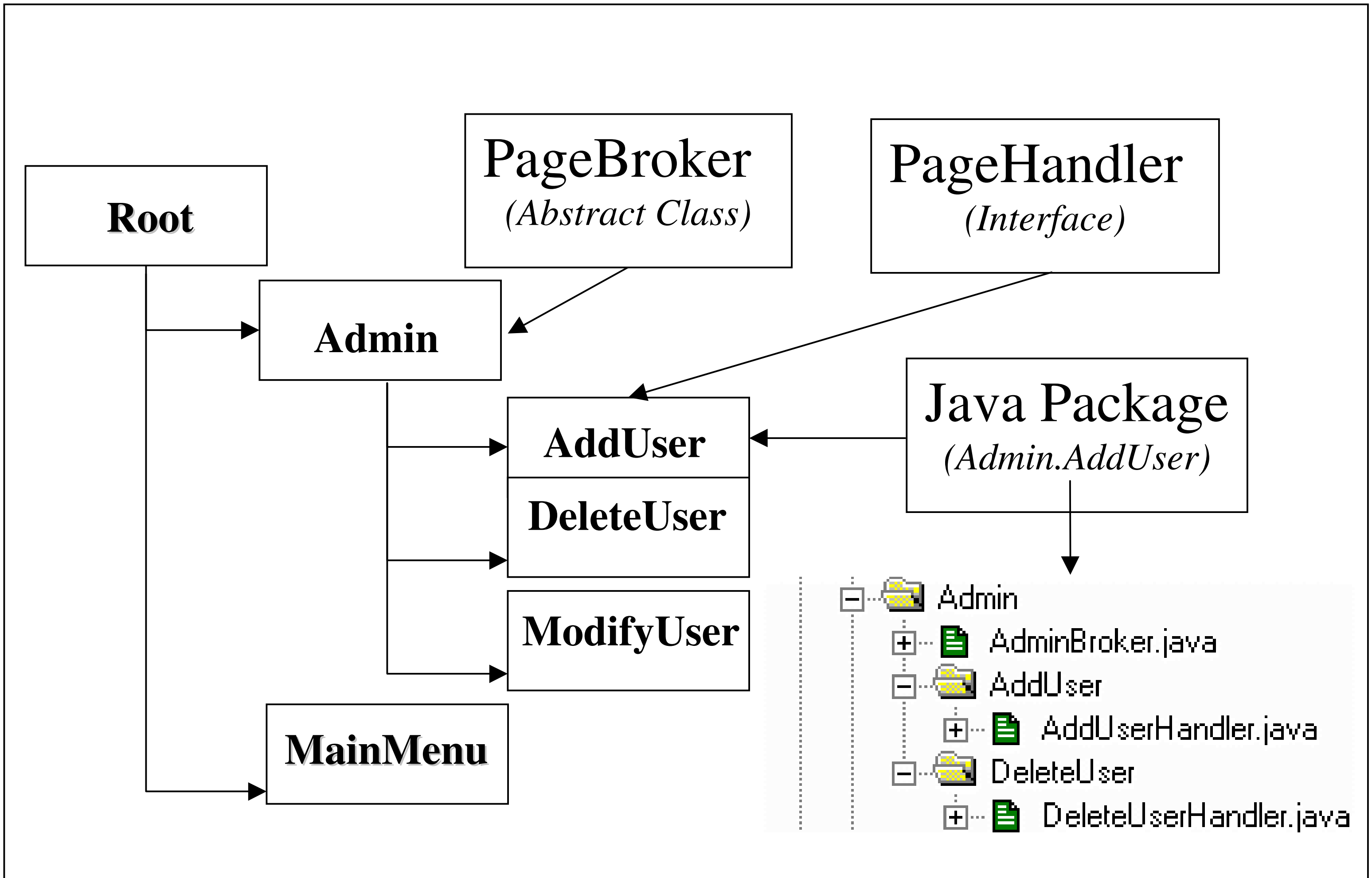


Figure 3

Template Manager

A commonly needed facility in web development is the ability to provide customized or dynamic HTML to the user. There are a number of ways this can be done. The simplest is actually making your program generate the HTML code using a series of print functions. By using a series of conditionals and print statements HTML can be constructed and sent to the user. This is obviously extremely unmanageable, and cannot be edited with normal web authoring tools. The Java servlet specification does not provide any facilities for creating dynamic HTML code. However a related specification, the Java Server Pages (JSP) specification, does describe a mechanism to create dynamic HTML.

The JSP specification describes a modified HTML language that includes special tags that allow Java commands to be inserted directly into HTML. These pages are then compiled into Beans, and are executed to produce HTML output. JSP files are only partially editable with an HTML editor. However, with JSP files you will still likely need to use print statements to generate some of the dynamic content. HTML editors will simply ignore the print statements stored in your JSP scripts, which may limit their usefulness. Until recently, the JSP standard had not yet been finished by Sun. Sun has been moving so slowly with the specification that several third-party vendors have implementations based on the incomplete specification. As a result, compatibility between JSP implementations may be an issue, as well as specification changes negatively affecting existing code. JSP is also not part of the servlet specification, and as a result may not be available in some or most servlet implementations.

As a result of the problems I found with JSP, I decided that a new dynamic HTML generating system needed to be created. I have created a high speed template system which is compatible with all HTML editing programs, including 4th generation tools. The system works on a simple substitution principle. A single special tag ($$.fieldname.) placed inside the HTML file denotes that it will be replaced with a dynamic value when the HTML page is rendered. When a PageBroker or PageHandler is initialized, it registers with the TemplateManager all the dynamic HTML files it will want to display. The template manager parses these files as they are registered to optimize rendering time. To render one of these files, the programmer would pass in a Hashtable of names and values to replace in the HTML file. For example, if the hashtable contains a key UserName, with a value of BobSmith, all instances of *$$.UserName.* in the HTML file will be replaced with BobSmith. This makes the HTML files used by the TemplateManager editable by HTML design programs. The *$$.UserName.* simply shows up as a place marker in the editor. The TemplateManager also parses the HTML form elements. Passing in a name-value pair for form elements will have the TemplateManager change the HTML to match your request. For example, passing in the name of ‘SupportRating’, with a value of 2, where ‘SupportRating’ is a radio button, the template system will automatically modify the radio button HTML code to mark 2 as the selected answer. This also works for input fields of most types, including select boxes, pulldown menus, textareas, text, hidden and password fields.

Logging Services

When writing an application, it is often helpful to be able to write to a log file, either to display debugging variables or to log errors. The Java servlet specification includes such a logging facility. Messages sent through this facility appear in the webserver event log. Unfortunately it does not have a useful facility for logging exceptions, or data types other than strings. As a result a Logging system is required. It serves as a layer between the servlet logging facility and the programmer. It has the ability to send its output to a separate log as well as the servlet log. Programmers can assign a priority to the log message as well. The programmer may want to assign high priority messages to a separate file or dispatch them via email. A very important function of this layer is to format exceptions. Although exceptions themselves can be converted to a string and printed to a logfile, their stack tracebacks cannot. To convert the traceback to a string a special extension had to be built, to format the output of the exception in a readable form. Now logging errors and exceptions is simple, and can be done from anywhere the Log manager is accessible.

Jini Services

If a single server became overloaded from incoming web requests it may become advantageous to spread the incoming web load over several systems. Many websites do this by having the hostname associated with the website (e.g www.yahoo.com) point to several IP addresses. Each of these IP addresses points to a separate computer, which has its own independent copy of the website. Usually the webservers will share a common database server, so content is the same. Client load is not spread out with any intelligence; each web user is given one of the IP addresses randomly by their DNS server. If one of the servers in the list needs to be replaced, or is to be removed unexpectedly, it may take anywhere from a week to a day for the DNS to update and reflect that change. For different priorities or distribution of load for a particular site function, a different mechinism must be used. An additional hostname would need to be issued for each of the different site functions. Although this may be seamless to integrate into a website, users may become confused when they are transferred to a different hostname as they travel through the site. As an alternative, web load can be distributed from the application level, instead of the network level. A finer level of control can be employed at the application level, such as prioritization of a particular class of users or requests and selective distribution, to distribute certain services to other servers. It is also possible to have a much more dynamic registration system, allowing servers to freely register and unregister as they wish.

Jini, a network service registration specification from Sun, is a close fit for this type of registration system. Each of the remote nodes that want to displace load off the main server announce themselves as a Jini service. The main server will then contact these nodes and start distributing requests from the web to them. The main node then only acts as a proxy, and sends the requests to the remote nodes registered to accept service requests. Then return the remote nodes response back to the webserver. The main node may use its own judgement as to what service is best suited to receive each incoming web request. Since we are using a modularized web structure, by breaking a whole web application into modules, each service may choose to only accept requests for certain modules.

Node Manager

There needs to be a service that is responsible for keeping track of the remote nodes currently available for requests. When the page manager is asked to service a request, it will ask the node manager to return a pointer to the most suitable candidate. For speed reasons, the Node Manager will not be querying the Jini lookup services for locating a suitable candidate. The node manager will need to keep a cached copy of the available services. The node manager will also need to accept notification from the page manager if a pointer is discovered to be no longer valid.

Node scout for Jini services

The node scout is an active thread inside of the node manager whose purpose is to scan the Jini lookup services looking for remote nodes that may have just registered themselves, or remote nodes that have unregistered themselves. The node scout can be triggered explicitly from the node manager, such as when a remote node is known to have unexpectedly disappeared. The node scout will also make time-based periodic scans of the Jini lookup services to look for node registration changes.

Real World Performance

The system uses a Jini based load balancing system in an attempt to decrease average response time to the web client, by spreading request load to multiple eligible systems. We measure the effectiveness of this load balancing system by using some simple benchmarking tools. Ultimately it is difficult to simulate real world usage, however we can very easily simulate concurrent requests. It is important to understand what is happening for each simulated request, to understand where the work is being done.

Request Trace

All client web requests come into the head-end node. The head-end node examines what PageHandler needs to handle the incoming request, and looks at the list of eligible nodes serving that PageHandler. The head-end node chooses the best node to handle the request, and sends the session and web-request to that node for processing. The head-end node waits for the response to be returned, and sends it to the web client. A graphical representation of this flow can be seen in Figure 4. It is important to understand that communication with the remote nodes uses

Java's RMI facilities. This requires that all objects sent between the head-end node and the remote nodes be serialized prior to transmission.

When benchmarking the system, the same exact web request was made of the web application. This request holds no state from the prior request, and a new session is created for each request. For each web request sent, a non-distributed server must do the following to service each request:

1. Issue a new session
2. Query the database, retrieve 10 article headlines, one "Quote of the Day" and one question with answers for the user poll
3. Use the dynamic html template system to build a webpage with the above database objects

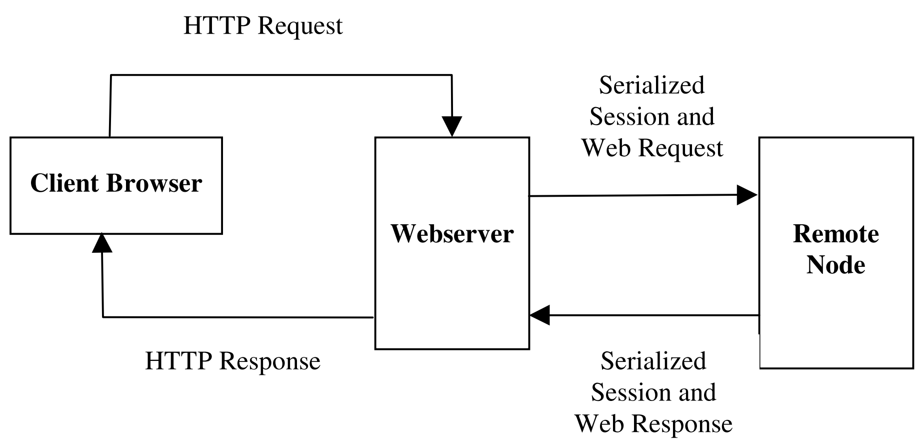


Figure 4 - Request

Load Distribution Algorithm

The head-end node is responsible for determining the best candidate for an incoming request. To do this the head-end node must have a speed ranking of all the remote nodes eligible for a request. It acquires this rating by periodically requesting a speed rating from each node. I call this the JavaBogoMips rating, a spin off of the Linux BogoMips rating. This JBM rating is the number of milliseconds it takes to execute a sequence of operations, so smaller is better. The test operations yield the thread several times to also get a reading on how many other threads are in the ready queue. The head-end node averages the minimum JBM rating (i.e. the best time), with the node's most recent reading. The head-end node also knows how many requests it has passed to each remote node, and how many are still outstanding.   It multiplies the speed rating of the machine by a weight for each outstanding request. Pseudo code implementing the relative speed rating follows:

```
public float getRelativeBogo() {
  float avgBogo = (minBogo+lastBogo) / 2;
  if (numActiveThreads>0) {
    avgBogo=avgBogo + (minBogo * (numActiveThreads+1)*4);
  }
  return avgBogo;
}
```

This accouts for a system getting slower as additional concurrently processing threads are added (i.e. additional load). Although a linear function used can approximate the speed of the remote machine, I suspect an exponential function would provide a more accurate simulation. I suspect this may be the case because the average response time of the web system does not degrade in a linear fashion as additional concurrent threads are added. A considerable amount of optimization of this estimation function would allow the head-end node to less periodically poll the remote nodes for their speed and more accurately estimate the speed of a remote system.

Benchmark Results

To simulate additional load, I increase the number of threads concurrently requesting the web application. There is no delay between requests made by the threads. This means that with no load distribution active, and 10 request threads, there should be approximately 10 processes/threads in the kernel ready queue awaiting service. In practice this does not end up being true, since some time is spent doing I/O, transferring the resulting HTML back to the thread.

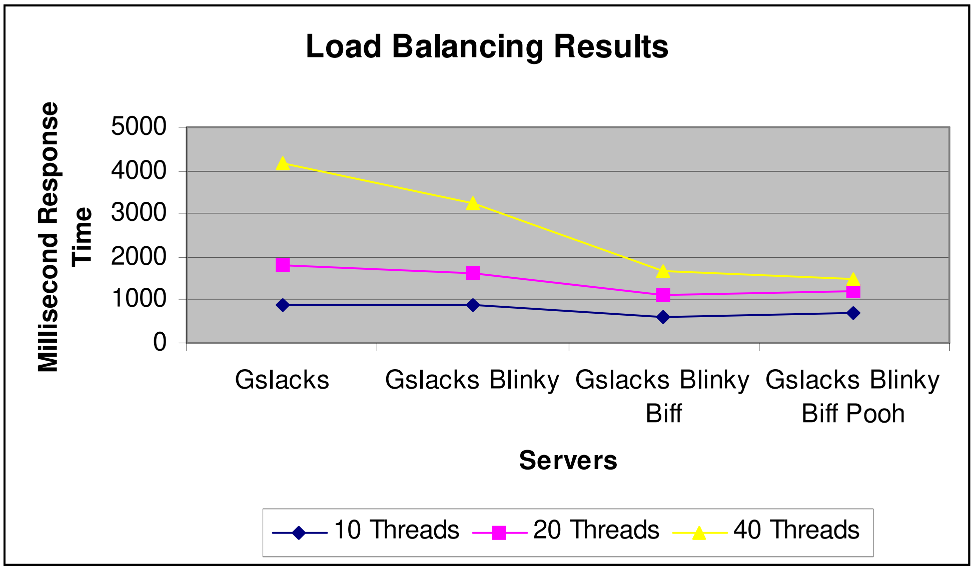


Figure 5 - Load Balancing Results

Before adding additional nodes, I felt it was very important to get a feel for the average response time with no nodes added. Gslacks served as the head-end node, running the database server, webserver, and Jini lookup services. As can be seen in figure 5, with 10, 20 and 40 concurrent requests and no remote nodes, the web application averaged 884ms, 1795ms, and 4165ms response time respectively.

By adding the first remote node, blinky, our average response time went down by 22% for 40 requests, and was reduced by a smaller percentage on fewer concurrent requests. The fact that there was any speedup at all comes as a slight surprise. The load distribution algorithm does allow the head-end node to take any web load. This means, by adding blinky, we have drastically changed what is occurring inside the web system. Before blinky was added, gslacks, the head-end node, was fielding all the incoming web requests, and also doing the processing locally. Now, with blinky added, gslacks is fielding all incoming requests, but not doing any web processing. Gslacks is transmitting all incoming requests to blinky for processing. Since web processing takes considerably more CPU time than simply proxying requests to nodes, blinky is taking an unfair load in this situation. This ends up being less then ideal, since we now have twice the computing power, but the system is still relying on only a single machine for web processing. I suspect the fact that there was any speedup at all in this case must only be due to separating the database and webserver from where the web processing threads are running. Although it is important to keep the head-end node as unloaded as possible, the work distribution in this case is clearly unfair.

Adding a second remote node spreads the web work over two nodes. For 40 threads this gives us a 60% decrease in average response time over no load balancing. We can also see that there is a smaller decrease percentage for fewer threads. It would have been an excellent test to add additional nodes to the system, and examine how the downward trend looks. After adding sufficient nodes, I suspect the downward trend would level out, as the head-end node would reach its brokering capacity. Unfortunately there were insufficient computers available to conduct such an experiment.

Adding pooh actually increased the average response time for 10 and 20 threads. This can be explained because pooh is a fairly slow machine, a PII-266. The other machines, gslacks, blinky and biff, will be able to execute the code required to serve the webpage faster then pooh, at least twice as fast. In the 10 and 20 threads case, even though gslacks, blinky and biff already have threads running on them, they are still producing results faster than pooh in an unloaded state. Although it is not shown in the graph, only 1 request at a time was being sent to pooh. Pooh was never allowed to concurrently process requests in the 10-20 thread cases. What this tells us is the threshold weight used to set how much slower a machine is running due to the number of running threads is too high. Simply put, pooh should not have been allowed to handle any requests for the 10 and 20 requests tests. Pooh was simply too slow to aid in bringing down the average. Pooh was also the only machine not located on the CMPSCI LAN. Pooh was located in the LGRT, and the added delay in communication may have adversely affected the speed of database queries, as well as the speed of communications with the head-end node. The load distribution algorithm does not take non-processor related delays into account.

Other Speed Considerations

Before adding additional hardware to solve a speed problem, it's important to examine simpler and more cost effective solutions. All of the components in this system, with the exception of the database server, run in a Java Virtual Machine. An increase in the efficiency of the JVM would increase the speed of almost all aspects of the system.

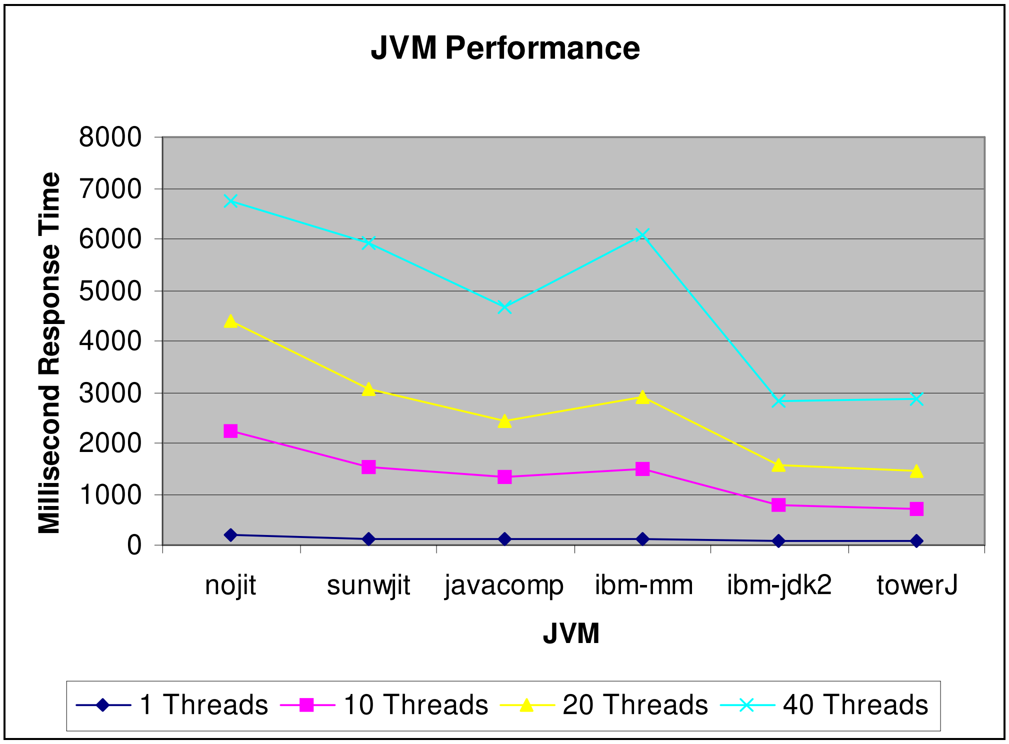


Figure 6 - JVM Performance

By trying different JVMs, we discover we could potentially halve our response time. “nojit” is the Blackdown port of the Sun JVM with no Just In Time compiler activated. “sunwjit” is the Blackdown port of the SUN JIT compiler plugged into the Blackdown JVM. “javacomp” is the Inprise JIT compiler plugged into the Blackdown JVM. “ibm-mm” is IBM’s 1.1.8 JVM. Ibm’s JVM is a mixed-mode VM which examines the execution flow of the program prior to optimizing and compiling the code to native instructions. The IBM 1.1.8 VM uses a different native threads implementation than the Blackdown JVM’s, and starts heavily degrading under heavy thread load. The IBM 1.1.8 JVM in other tests outperformed even the Inprise JIT in single threaded applications. “ibm-jdk2” is a beta release of the IBM 1.3 JVM. IBM made a considerable amount of progress since their 1.1.8 VM, especially under heavy thread load. “TowerJ” is a commercial product which converts compiled Java class files to pure C++ code. It then runs a C compiler over the produced code and produces a native binary. Although TowerJ is the fastest JVM tested here, TowerJ is very expensive, and inconvenient. The development "starter kit" costs more then a compact car, and deployment licensing is even more expensive. TowerJ requires the entire application to be recompiled to C++ code and then to native code when a single class file is updated. This process can take as long as a half an hour on a medium classed machine. TowerJ would be an excellent deployment JVM, but a poor JVM to use during development.

The load balancing benchmarks reported in the previous section were run using "sunwjit", the Blackdown port of the sun JIT compiler. This suggests that it should be possible to nearly double throughput by switching to a superior JVM such as the ibm-jdk2.

Limitations

If a website is under sufficient load that multiple machines are being dedicated to serving it through this load balancing scheme, redundancy is most likely also an important consideration. In this respect, this method of distributing load has some very serious limitations. First, there is a single point of failure. In the event of a head-end node failure, the remote-nodes would have no way of handling web load, and would effectively become zombies. Currently, the remote-nodes are capable of communicating with multiple head-end nodes. Unfortunately, head-end nodes do not have the ability to correctly function with other head-end nodes. This is because adding a second head-end node causes problems with session management. Currently all sessions are stored in memory on the head-end node. If two head-end nodes were used, and a rotating DNS entry was made to spread user requests over the two nodes, a user's session would only exist on one of the head-end nodes. If a user connected to one head-end node, then connected through the other head-end node, the system would be unable to access the user's session. If multiple head-end nodes were to be used, sessions would need to be stored in a common location such as a redundant database. This severely limits the convenience of the current session system, which allows objects of any type to be placed in a session, not just objects suitable for database storage.

Another serious limitation of having a single head-end node is that you have a single point which, if overloaded, degrades the performance of the entire system. Unless a second head-end node is introduced, the system would never be able to use the additional resources of remote nodes once the head-end node reaches its maximum number of requests per second.

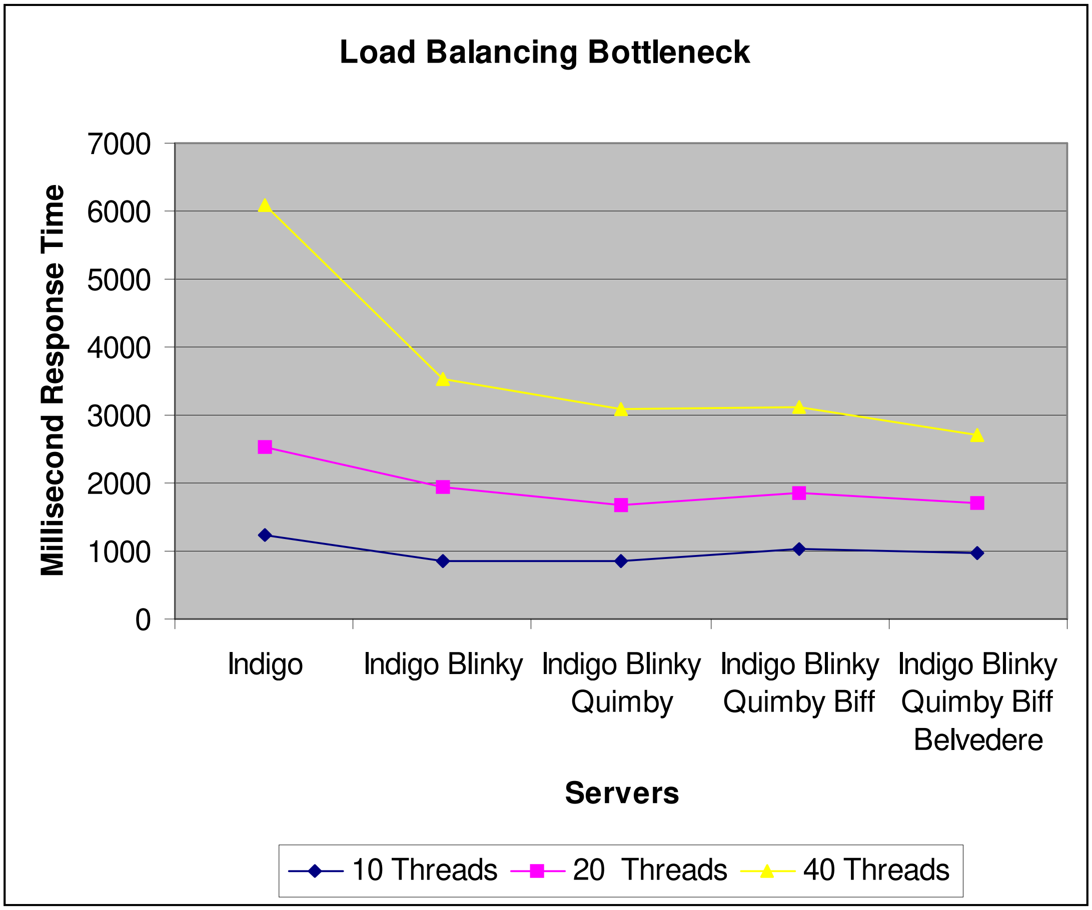


Figure 7 - Load Balancing Bottleneck

Figure 7 shows the results of an overload situation. In this case indigo is serving as the head-end node. The remote nodes blinky, quimby and biff are approximately five times as fast as the head-end node indigo. It becomes clear that adding additional nodes has little effect on decreasing the average response time of the system. Indigo is simply unable to proxy requests to its remote nodes fast enough to take advantage of them. The remote nodes in this case are spending a considerable amount of CPU time idle.

Different Philosophies

There are a number of different ways we could leverage having additional systems to aid the speed of the system. One way is to broker entire web requests out to the additional systems. This is the method implemented currently in the system. One large advantage of this load distribution scheme is it requires no prior planning to implement. No modification to the web site code should be necessary. If a web application is written using this framework, it is usually as easy as flipping a switch to start the web application using the load distribution scheme.

Unfortunately if only one web request comes in concurrently, there will be no increase in speed, regardless of how many systems are available.

Another methodology to increase throughput is to try to parallelize a single web request as much as possible. Unfortunately this can only be done by the programmers using the framework to actively program their code in a parallel manner. We could create an interface similar to a Thread, perhaps called a DistributableThread. When started, the framework would scan to see if there are eligible remote nodes capable of handling the request. If there are none, it would simply fire up a standard local thread, and execute the code. If there are eligible nodes, the DistributeableThread would be started on the remote node. When the programmer starts one of these threads, it would return immediately so it can continue execution in parallel. This means the programmer would at some point need to check and retrieve the results of the DistributableThread. This would work like a mailbox system. There would be non-blocking and blocking methods to check to see if the DistributableThread has completed execution. There would also be a blocking method to retrieve the return result. This has the potential to speed up the case where only a single concurrent request is coming into the system. Unfortunately it would only do so if the program is parallellizable, and is written as such.

Conclusion

This project makes an attempt to create a framework that takes advantage of the technology available today, Java servlets. The facilities provided to a programmer using the framework simplify and structure the creation of their web application. A mechanism is also provided for increasing the web system performance through the use of additional machines. Although this mechanism is functional, it is still in its early stages. Several major limitations are present in the current system. The algorithms used are also not optimized for maximum efficiency. Future enhancements to the framework may even ultimately make use of additional machines in an entirely different way for increased performance. The framework created will continue to evolve as technology and industry requirements continue to change.